软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2011,22(10):2523−2537 [doi: 10.3724/SP.J.1001.2011.04048]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

一种连续数据保护系统的快照方法*

李 虓, 谭毓安, 李元章+

(北京理工大学 计算机学院,北京 100081)

Snapshot Method for Continuous Data Protection Systems

LI Xiao, TAN Yu-An, LI Yuan-Zhang+

(School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China)

+ Corresponding author: E-mail: popular@bit.edu.cn

Li X, Tan YA, Li YZ. Snapshot method for continuous data protection systems. *Journal of Software*, 2011,22(10):2523−2537. http://www.jos.org.cn/1000-9825/4048.htm

Abstract: This paper proposes a new snapshot method for continuous data protection (CDP) system that considers the disability of taking large amount of snapshots in traditional CDP systems. The snapshot method (Convex Point SNAPshot, CSNAP) is based on the concept of convex point set. After the data structure of CSNAP and introduced the concept of convex point based on the pointers in the data structure have been discussed, the study analyzes the properties of convex point set and proposed CSNAP algorithms. An enhanced CSNAP method by introducing the concept of retro-cost is also proposed. Finally, the study uses a typical workload and random generated trace data to test CSNAP method. The experimental results show that at average CSNAP takes less than 10% storage space of traditional snapshot method.

Key words: continuous data protection; snapshot; convex point; concave point; retro-cost

摘 要: 针对传统连续数据保护技术中使用的快照存储技术难以高效存储足够多快照的问题,提出了一种新的快照存储技术,即基于凸点集合的快照存储技术(convex point SNAPshot,简称 CSNAP).在讨论了 CSNAP 技术使用的数据结构,并引进了在数据结构中加入指针构成的凸点概念的基础上,分析了凸点集合具有的基本性质,并给出了利用凸点集合储存快照的相关算法,同时还给出了一种使用逆行代价改进 CSNAP 的方法.最后通过在实际工作负载和随机合成工作负载下的模拟实验说明了,在平均情况下,使用改进的 CSNAP 技术,可以将快照占用的储存空间减少到传统方法的 10%以下.

关键词: 连续数据保护;快照;凸点;凹点;逆行代价

中图法分类号: TP316 **文献标识码**: A

随着信息技术的不断发展,数据信息已在越来越多的企业和组织中扮演着至关重要的角色.正是由于能够对海量信息进行自动化处理和智能分析,才使得各种组织能够正常而高效地运转,大型企业能够提高利润率从而获得丰厚的回报.但是,数据信息同时也正面临着各种潜在的风险,例如人为或者技术上的错误、自然灾害,甚至恐怖袭击.根据统计数据显示,硬件故障成为最大的风险来源,在所有数据损失中,有 44%源自硬件故障.用户误操作导致的数据损失占 32%,软件错误占 14%,而计算机病毒和自然灾害导致的数据损失分别占总数的 7%

* 收稿时间: 2010-05-10; 修改时间: 2010-12-09; 定稿时间: 2011-04-28

和 3%[1].

为了保护重要的数据信息,可以采用数据备份技术,在数据不可用时能从备份介质中恢复数据.但在恢复数据时,两次备份之间的数据会丢失.对于一些关键的信息系统,所能容忍的数据丢失量即恢复点目标(recovery point object,简称 RPO)要尽可能地低,需要应用连续数据保护(continuous data protection,简称 CDP)技术[2].

CDP 将更改过的所有数据按时间顺序保存下来,每次写操作都会生成带有时间戳的数据版本,在恢复数据时能够获取任意一个时间点的数据状态.根据实现层次的不同,可以把 CDP 分成 3 类:应用级连续数据保护、文件级连续数据保护以及块级连续数据保护.块级 CDP 将一个磁盘即逻辑单元(logical unit,简称 LUN)划分为固定大小的数据块,以数据块为单位记录数据变化,特点是与应用的耦合比较低,性能和效率优于文件级 CDP[3,4].

当需要恢复数据到某个指定的时间点时,CDP 系统可以从开始时间点遍历所有已经记录的修改过的数据块,直到指定的时间点,以恢复该时间点的数据状态,即获得该时间点的磁盘的每一个逻辑块地址(logical block address,简称 LBA)与 CDP 所记录的数据块之间的映射关系[5−8].

为了缩短遍历时间,通常需要在 CDP 系统中定期地产生数据快照,将某一时刻的映射关系存储起来.当恢复快照时刻的数据时,就可以直接获得映射关系,而无需再遍历 CDP 日志;要恢复数据到其他时间点时,从该时间点之前的那一次快照开始,遍历 CDP 日志直到该时间点即可获得映射关系,而无需从开始时间点遍历[9−11].

数据快照越密集,数据恢复所需的时间就越短,能够满足更低的数据恢复时间目标(recovery time object,简称 RTO).然而,CDP 系统生成快照需要付出存储空间的代价,生成快照时需要保存每一个块地址(LBA)到日志数据块之间的映射[12−14].

本文提出了一种在块级 CDP 中高效存储快照的方法,使用少量的存储空间就可以保存快照时刻的映射关系,以支持更密集的数据快照,有利于缩短数据恢复时间.

1 连续数据保护系统的传统快照方法

在一个具有最基本功能的传统连续数据保护(CDP)系统中,对磁盘数据块的每一个写操作,CDP 系统中都有一个记录该操作的 CDP 元数据结构.CDP 元数据包含数据块的块号和时间戳.当 CDP 系统捕获到对磁盘数据块的一个写操作时,生成一个元数据结构.CDP 系统在日志数据区中记录数据块内容,在日志元数据区中记录数据块元数据.同时,CDP 系统还需要实时维护一个映射表,表中每一项记录一个磁盘数据块的最后一次写入操作元数据在 CDP 日志中的位置,并且使用一个特殊值表示从开始记录 CDP 日志以来尚未修改的数据块.

当 CDP 系统需要保存当前时刻的快照时,将系统当前时刻的映射表内容作为快照元数据保存在磁盘上.当需要恢复已经保存在磁盘上的快照时,从快照元数据中获取快照时刻的映射表内容,再根据映射表中指定的每个数据块最后一次修改在 CDP 日志中的位置,即可恢复当时的磁盘数据内容.

当需要恢复快照之外的其他时间点数据时,首先恢复到该时间点之前的一次快照,获得快照时刻的映射关系,再遍历快照时刻到该时间点之间的 CDP 元数据,根据元数据内容更新映射关系.

1.1 数据结构与约定

假设这样一个基本 CDP 系统需要记录一个逻辑单元(LUN)的磁盘块地址 BLK_{start} 到 BLK_{end} 共 $BLKNR=BLK_{end}-BLK_{start}+1$ 个数据块(通常为 4KB 大小)的日志,则由上面的描述,可以抽象出以下数据结构:

(1) 需要一个保存着 BLK_{start} 到 BLK_{end} 索引的日志当前状态表 $BLKTAB$,对表中每一项索引 BLK_n,表中保存 $last(BLK_n)$,即 BLK_n 最后一次写入的位置.为了讨论方便,使用下标表示某一个时间点的状态表内容,例如 $BLKTAB_t$ 表示 t 时刻的状态表,$last_t(BLK_n)$表示 t 时刻块号为 BLK_n 的数据块的最新修改位置;

(2) 对于第 k 个写入的数据块 WB_k,按时间顺序保存一个对应的元数据结构 MT_k,其中包括 $timestamp(MT_k)$ 和 $blockno(MT_k)$,分别表示数据块的写入时间戳和数据块的 LBA;

(3) 为了实现秒级恢复,可以设置一个秒级索引 $INDEX$,其中每一项指向对应时刻第 1 个写入数据块在 CDP 日志中的元数据位置.

为了下文进一步讨论的方便,我们有以下约定:

约定 1. 储存 $timestamp(MT_k)$时使用的是顺序索引号,而非实际时间,所以有,

$$timestamp(MT_t)=t.$$

因为每一个写入 CDP 日志的块 MT_t 被分配了唯一的顺序索引号 t 作为时间戳,所以在写入块和时间戳之间存在一一对应关系.顺序索引号由 1 开始编号,因此当下文提及第 k 个写入块 MT_k 时,其写入时刻即为 k.

1.2 传统快照方法的限制

由上文可以看出,在 CDP 系统的传统快照方法中,快照本身的元数据将占用大量的存储空间.例如,当需要对一个 512GB 的逻辑卷记录 CDP 日志时,假设采用 4KB 的数据块大小进行记录,则我们可以进行以下计算:

CDP 日志的当前状态表 *BLKTAB* 需要保存每一个数据块地址的最后一次修改在 CDP 日志中的位置.如果使用 64 位的逻辑块地址(LBA),则对于每一个 BLK_n,字段 $last(BLK_n)$需要 8 个字节.一个 512GB 的逻辑卷的块数为 128M(512GB/4KB=128M),因此快照表的项数也是 128M,所以状态表 *BLKTAB* 的大小为 1GB.如果在这个 CDP 系统中需要保存一次快照,即需要 1GB 的存储空间.

可见,这种大小的快照在读取和写入的过程中都将产生较大的开销,而且不适合以较高的频率保存快照.而在一个 CDP 系统中,如果没有足够多的快照,则在进行任意时间点恢复时,就需要从最近的快照开始遍历 CDP 日志,这同样将导致恢复过程需要较长的时间.下文将要讨论的即是一种通过减少单个快照容量从而实现高频率快照并且提高快照存取效率的快照方法.

2 连续数据保护系统的 CSNAP 快照方法

由上文对 CDP 系统中传统快照方法的说明可以看出,导致快照数据量巨大的原因在于,需要针对被保护磁盘的每一个逻辑地址记录其最后一次修改数据在 CDP 日志中的位置.为了减少快照储存空间,一种自然的想法是减少需要保存的逻辑地址数量,只保存相对关键的逻辑地址的最后修改位置.根据这种想法,我们提出了 CDP 系统的 CSNAP(convex point SNAPshot)快照方法.

CSNAP 方法将被保护磁盘某一时刻的 LBA 地址空间分为 3 个部分,即凸点地址部分、凹点地址部分以及其他一般地址部分.通过在 CDP 日志元数据结构中加入相关地址间的链接信息,使得在保存快照时只需保存 LBA 地址空间中的凸点地址部分,从而减少快照数据的储存空间,提高快照的存取速度.

下面就从凸点的基本定义开始,说明连续数据保护系统的 CSNAP 快照方法.

2.1 定义与基本事实

2.1.1 数据结构扩展

为了实现 CSNAP 快照方法,需要对传统快照方法中的数据结构进行扩展:

(1) 需要一个保存着当前时刻“凸点”集合的平衡二叉树结构:*CONVEX*.*CONVEX* 按照凸点块号大小顺序索引.为了讨论方便,我们使用下标表示某个特定时间点的“凸点”集合,例如,$CONVEX_t$ 表示在 t 时刻的凸点集合;

(2) 对于第 k 个写入块 WB_k,扩展传统快照方法中的元数据结构 MT_k,除去已包含的 $timestamp(MT_k)$, $blockno(MT_k)$外,增加前向链接 $nextwrite(MT_k)$、上行链接 $prevblock(MT_k)$、下行链接 $nextblock(MT_k)$ 这 3 个字段.假设 MT_k 对应的块号为 L,则 3 个字段分别表示 L 的下一次修改位置、L–1 的当前最新修改位置(即 $last_k(blockno(MT_k)-1)$)及 L+1 的当前最新修改位置(即 $last_k(blockno(MT_k)+1)$).

第 1 项中凸点的定义将在下文中给出.同时,第 2 项中新增加的 3 个字段的作用将会在相应的算法中说明.

2.1.2 凸点的定义及相关概念

定义 1. 每个时刻 t 确定了一个快照 $SNAPSHOT_t$,即由 BLK_{start} 到 BLK_{end} 的所有块号及其在 t 时刻的最新改写组成的有序对构成的集合:$\{(BLK_{start},last_t(BLK_{start})),\ldots,(BLK_{end},last_t(BLK_{end}))\}$.

上面对快照的概念给出一个定义,这和上文中传统快照的概念是一致的,即快照是由状态表 *BLKTAB* 的索引及其对应 *last* 字段中的数据构成的集合.下面,为说明方便,认为 MT_k 属于 $SNAPSHOT_t$ 就是指($blockno(MT_k)$,

MT_k)属于 $SNAPSHOT_t$.

定义 2. MT_k称为 t 时刻的一个凸点,如果 MT_k属于 $SNAPSHOT_t$,且 MT_k的上下行连接均属于 $SNAPSHOT_t$:

$MT_k \in CONVEX_t$ **iff** $last_t(blockno(MT_k))=MT_k$ **and**

$last_t(blockno(MT_k)-1)=prevblock(MT_k), last_t(blockno(MT_k)+1)=nextblock(MT_k)$

凸点的概念来源于函数的极值点.因为快照中包括了所有块号的当前最新修改位置,每个位置对应一个修改时刻.如果将修改时刻看作块号的函数,则凸点对应于函数的极大值点.即在凸点块号的某个邻域内,所有其他块号的最新修改时刻早于凸点的最新修改时刻.类似地,我们可以定义凹点:

定义 3. MT_k称为 t 时刻的一个凹点,如果 MT_k属于 $SNAPSHOT_t$,且 MT_k的上下行连接均不属于 $SNAPSHOT_t$:

$MT_k \in CONCAVE_t$ **iff** $last_t(blockno(MT_k))=MT_k$ **and**

$last_t(blockno(MT_k)-1)\neq prevblock(MT_k), last_t(blockno(MT_k)+1)\neq nextblock(MT_k)$

这里,集合 $CONCAVE$ 只用于说明,实际并不需要一个对应的数据结构.

根据定义可知,在不需要 $last_t$ 的情况下,可以在常数时间判定一个块 MT_k 在 t 时刻的凹凸性:

function $convex(MT_k,t)$:

if $before(MT_t,nextwrite(MT_k))$ **and**

$before(MT_t,nextwrite(prevblock(MT_k)))$ **and**

$before(MT_t,nextwrite(nextblock(MT_k)))$ **then**

reture true

else

reture false

类似的方法可以用于判断凹点,其中,*before* 用于比较两个元数据块的写入时刻:

function $before(MT_a,MT_b)$:

if $MT_a=nil$ **then**

reture true

if $MT_b=nil$ **then**

reture false

if $timestamp(MT_a)<timestamp(MT_b)$ **then**

reture true

else reture false

2.1.3 有关快照和凸点的基本性质

根据以上给出的定义和概念,我们可以得到以下基本事实:

事实 1. MT_t为 t 时刻的一个凸点.

事实 2. 任意两个相邻的凸点之间存在一个凹点,任意两个相邻凹点之间存在一个凸点.

事实 3. 若 MT_t上(下)相邻块(LBA 相邻)在 t–1 时刻是凸点,则在 t 时刻不再是凸点.

事实 4. 任意时刻 t 的快照由 $last_t$ 决定.即到该时刻为止,所有块的最新修改.

事实 5. 若 $MT_t\neq nil, before(prevblock(MT_t),MT_t)$=true,

若 $MT_t\neq nil, before(nextblock(MT_t),MT_t)$=true.

事实 6. 若 $nextwrite(prevblock(MT_t))\neq nil, before(MT_t,nextwrite(prevblock(MT_t)))$=ture,

若 $nextwrite(nextblock(MT_t))\neq nil, before(MT_t,nextwrite(nextblock(MT_t)))$=ture.

事实 7. 由 t 时刻的某个凸点开始,沿上(下)行连接进行遍历,直到 $prevblock(MT_k)$($nextblock(MT_k)$)不属于 $SNAPSHOT_t$,则 MT_k为一个凹点.

事实 8. $\forall MT_k\in SNAPSHOT_t, MT_k\neq MT_t \Rightarrow before(MT_k,MT_t)$=ture.

事实 9. 若 $MT_k\in CONVEX_t$,则 $prevblock(MT_k)\in SNAPSHOT_t$.

且 $nextblock(MT_k)\in SNAPSHOT_t$

2.2 基本CSNAP方法

有了以上定义和事实,我们就可以得到下面的基本 CSNAP 快照算法.

2.2.1 CDP 系统日志记录算法

插入一个块号为 $BLK_{new}(BLK_{start}\leqslant BLK_{new}\leqslant BLK_{end})$的数据块到 CDP 日志中的算法:

1. 将数据块的内容 WB_k 写入到 CDP 日志数据区中;
2. 构造数据块的元数据结构 MT_k:
   a) 写入时间戳:$timestamp(MT_k)\leftarrow k$;
   b) 写入数据块块号:$blockno(MT_k)\leftarrow BLK_{new}$;
   c) 更新上次改写的前向链接:$nextwrite(last(BLK_{new}))\leftarrow MT_k$;
   d) 本次写入的前向链接置空:$nextwrite(BLK_{new})\leftarrow nil$;
   e) 构造上行链接:

      if $BLK_{new}\neq BLK_{start}$ **then**

         $prevblock(MT_k)\leftarrow last(BLK_{new}-1)$

      else $prevblock(MT_k)\leftarrow nil$;

   f) 构造下行链接:

      if $BLK_{new}\neq BLK_{end}$ **then**

         $nextblock(MT_k)\leftarrow last(BLK_{new}+1)$

      else $nextblock(MT_k)\leftarrow nil$;
3. 更新日志当前状态索引表 *BLKTAB*:
   a) 更新末次写入位置:$last(BLK_{new})\leftarrow MT_k$;
4. 更新当前凸点集合:
   a) 将本次写入插入凸点集合:

      if $MT_k\notin CONVEX$ **then** $CONVEX\leftarrow CONVEX\cup\{MT_k\}$;

   b) 删除上方可能凸点:

      if $prevblock(MT_k)\in CONVEX$ **then** $CONVEX\leftarrow CONVEX-\{prevblock(MT_k)\}$;

   c) 删除下方可能凸点:

      if $nextblock(MT_k)\in CONVEX$ **then** $CONVEX\leftarrow CONVEX-\{nextblock(MT_k)\}$.

2.2.2 CDP 系统生成快照算法

保存 *t* 时刻的快照:

foreach MT_i **in** $CONVEX_t$ **do save** MT_i **in** $SNAPSHOT_t$

即,将凸点集中的数据块元数据在 CDP 日志元数据区的位置存入快照,而不必将元数据本身的内容存入.

2.2.3 CDP 系统恢复快照算法

恢复 *t* 时刻的快照,即重建 *t* 时刻的 *BLKTAB* 中的 *last* 列到一个临时表 *TAB*,*TAB* 中的每一个块的索引指针首先被初始化为 *nil*,表示该数据块在原盘而非 CDP 中(即开始 CDP 后,该块尚未改写):

1. 对快照中的每一个凸点执行下面的步骤 2~步骤 4:
2. 将该凸点的值加入 *TAB*;
3. 上行查找,将途经的每个 MT_i 加入 *TAB*;
4. 下行查找.将途经的每个 MT_i 加入 *TAB*:

   foreach MT_i **in** $SNAPSHOT_t$ **do**

      $last_{TAB}(blockno(MT_i))\leftarrow MT_i$

      $MT_u\leftarrow MT_i$

while $prevblock(MT_u)\neq nil$ **do**
  if $before(MT_t,nextwrite(prevblock(MT_u)))$ **then**
    $MT_u \leftarrow prevblock(MT_u)$
    $last_{TAB}(blockno(MT_u)) \leftarrow MT_u$
  else break
$MT_u \leftarrow MT_i$
while $nextblock(MT_u)\neq nil$ **do**
  if $before(MT_t,nextwrite(nextblock(MT_u)))$ **then**
    $MT_u \leftarrow nextblock(MT_u)$
    $last_{TAB}(blockno(MT_u)) \leftarrow MT_u$
  else break

其中,$before(MT_a,MT_b)$用于判断两个块的元数据写入时间顺序,这可以通过比较它们在元数据日志中的先后位置而得到.

例如,在下面的图 1 中,假设即将写入 MT_{11}.此时凸点集 $CONVEX=\{MT_6,MT_8,MT_{10}\}$,在图中标为深色.首先将 MT_{11} 加入,于是 $CONVEX=\{MT_6,MT_8,MT_{10},MT_{11}\}$,由于 $prevblock(MT_{11})=MT_{10}$,$nextblock(MT_{11})=MT_6$,所以从 $CONVEX$ 中删除 MT_{10} 和 MT_6:$CONVEX=\{MT_8,MT_{11}\}$.若此时要保存快照,则只需保存两个块的元数据地址索引.

当要从时刻 11 的快照 $SNAPSHOT_{11}$ 中恢复映射关系时:从 $SNAPSHOT_{11}$ 中取出一个凸点 MT_{11},然后沿着 $prevblock$ 向上查找,找到 MT_{10},MT_9(图中方点线).由于 $prevblock(MT_9)=nil$,向上查找结束.接着由 MT_{11} 沿着 $nextblock$ 向下查找,找到 MT_6,MT_4(图中短划线).此时,由于 $nextblock(MT_4)=nil$,向下查找结束.接着取出下一个凸点 MT_8,继续上面步骤.

再如,要恢复时刻 10 的快照,此时快照中有 3 个凸点 $CONVEX=\{MT_6,MT_8,MT_{10}\}$.当由 MT_6 向下查找时,首先找到 MT_5,但 $nextwrite(nextblock(MT_5))=MT_6$,而 $MT_6<MT_{10}$(即 $before(MT_{10},MT_6)$=false),这说明 MT_5 是一个凹点,所以向下查找结束.

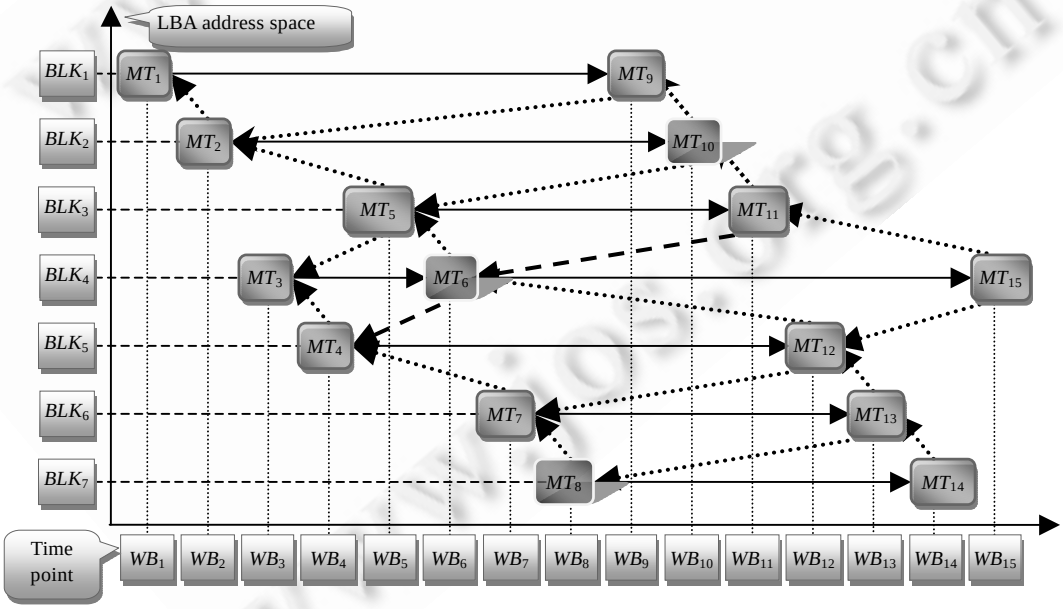


Fig.1 CSNAP algorithm example

图 1 CSNAP 算法示意图

2.3 改进CSNAP方法

使用“逆行代价”可以进一步改进算法,只需在 *BLKTAB* 中记录每个 LBA 块当前对应 *prevblock* 和 *nextblock* 的“上下逆行代价”:*prevcost*,*nextcost*.

比如在下面的图 2 中,MT_1 对应于 *prevblock* 的上逆行代价为 1,而 MT_2 为 2,MT_3 为 3.

这是因为,要从 *prevblock*(MT_1)=MT_4 找到 MT_1,需要沿着 *nextblock*(*prevblock*(MT_1))=MT_5 的前向连接搜索 1 步.同理,MT_2 需要 2 步,MT_3 需要 3 步.

利用逆行代价,在保存快照时可以评估哪些凸点可以不用保存而只需利用这个凸点上下两个凹点中的一个,在恢复快照时通过逆行搜索达到.

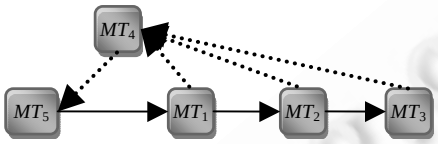


Fig.2 Retro-Cost example

图 2 逆行代价示意图

并且在记录逆行代价时,只需记录每个块最后修改时的逆行代价.如图 2 所示,当 MT_3 写入时,只需计算出 MT_3 的逆行代价代替 MT_2 的逆行代价(因为这个值只是用来评估某个当前凸点在保存快照时是否应该写入).这个值可以通过 MT_2 和 MT_4 得到:如果 MT_2<MT_4<MT_3(即 MT_4 的写入时刻在 MT_2 和 MT_3 之间,则 MT_3 的逆行代价为 1;如果 MT_4<MT_2<MT_3(图中情况),则 MT_3 的逆行代价为 MT_2 的逆行代价加 1.只可能出现这两种情况.

因此,在将一个块插入到 CDP 日志中时,只需在更新 *BLKTAB* 时在更新 *last* 字段之前增加常数时间的操作,就可以在 *BLKTAB* 中维护两个新的列:*prevcost* 和 *nextcost*:

a) 构造上逆行代价:

if BLK_{new}≠BLK_{start} **then**
  if *before*(*last*(BLK_{new}),*prevblock*(MT_k)) **then**
    prevcost(BLK_{new})←1
  else *prevcost*(BLK_{new})←*prevcost*(BLK_{new})+1
else *prevcost*(BLK_{new})←0

b) 构造下逆行代价:

if BLK_{new}≠BLK_{start} **then**
  if *before*(*last*(BLK_{new}),*nextblock*(MT_k)) **then**
    nextcost(BLK_{new})←1
  else *nextcost*(BLK_{new})←*nextcost*(BLK_{new})+1
else *nextcost*(BLK_{new})←0

其中,MT_k 为写入块的元数据,*blockno*(MT_k)=BLK_{new}.

在上面恢复 MT_{10} 的例子中,如果限制逆行代价的阈值为 1,则完全不需要保存任何凸点(MT_{10} 本身需要保存用于指示快照建立的位置).因为 MT_8 可以由 MT_4 向下逆行找到.首先,MT_4 找到 MT_7(因为 MT_{13} 已经在 MT_{10} 之后,所以到 MT_7 就结束第 1 次的逆行搜索).然后继续第 2 次逆行搜索就能找到 MT_8.

在评估凸点是否需要写入时,由 MT_4(凹点)到 MT_8(凸点)这条逆行路径的逆行代价,可以由累计每两点间的逆行代价(1+1),再平均到路径长度(2)来得到.为了简化算法的复杂性,可以只考虑由凹点到相邻凸点这种路径的逆行代价.虽然可能考虑凹点到隔开几个凸点的不相邻凸点的路径的逆行代价,可以获得更好的平均结果,但算法过于复杂.

在需要记录 *t* 时刻的快照时,首先遍历当前凸点集合,累积每个凸点到上下相邻凹点的路径逆行代价

prevpathcost 和 *nextpathcost*.事实 9 保证,不会出现除零错误.

foreach MT_i **in** $CONVEX_t$ **do**
  $MT_u \leftarrow MT_i$
  $pathlen \leftarrow 0$
  $pathcost \leftarrow 0$
  do
    if $prevblock(MT_u)=nil$ **and**
      $((blockno(MT_i)-pathlen=BLK_{start}$ **or** $last(blockno(MT_i)-pathlen-1)\neq nil)$ **then**
        break
    if $prevblock(MT_u)\neq nil$ **and** $nextwrite(prevblock(MT_u))\neq nil$ **then**
        break
    $pathcost \leftarrow pathcost+prevcost(blockno(MT_i)-pathlen)$
    $pathlen \leftarrow pathlen+1$
    $MT_u \leftarrow prevblock(MT_u)$
  while $MT_u \neq nil$
  $prevpathcost(MT_i) \leftarrow (blockno(MT_i)=BLK_{start})?\infty:pathcost/pathlen$
  $MT_u \leftarrow MT_i$
  $pathlen \leftarrow 0$
  $pathcost \leftarrow 0$
  do
    if $nextblock(MT_u)=nil$ **and**
      $((blockno(MT_i)+pathlen=BLK_{end}$ **or** $last(blockno(MT_i)+pathlen+1)\neq nil)$ **then**
        break
    if $nextblock(MT_u)\neq nil$ **and** $nextwrite(nextblock(MT_u))\neq nil$ **then**
        break
    $pathcost \leftarrow pathcost+nextcost(blockno(MT_i)+pathlen)$
    $pathlen \leftarrow pathlen+1$
    $MT_u \leftarrow nextblock(MT_u)$
  while $MT_u \neq nil$
  $nextpathcost(MT_i) \leftarrow (blockno(MT_i)=BLK_{end})?\infty:pathcost/pathlen$

由下面图 3 可以简单说明利用每个凸点的上下路径“逆行代价”简化凸点集,从而减小快照大小的方法.

图中第 1 行的 11 个圆点代表 $MT_a,\ldots,MT_k$,其中省略了相邻凸点之间的凹点.假设这是某个时刻快照中的所有凸点,则原始算法需要保存所有这 11 个凸点.

现在使用每两个相邻 *BLK* 之间的当前逆行代价,累积每个凹点到它的两个相邻凸点的路径逆行代价.图中第 2 行、第 3 行加入的实线箭头表示两个凸点间的凹点可以沿着箭头方向,在预设的平均路径逆行代价阈值范围内搜索到箭头指向的凸点.换言之,箭头指向的凸点在箭头侧的平均路径逆行代价在阈值范围内.比如在第 2 行中有 $MT_a \rightarrow MT_b$,就表示 MT_a,MT_b 间的凹点可以逆行搜索到 MT_b,因此可以不用在快照中储存 MT_b,而是由 MT_a 沿着下行连接达到 MT_a,MT_b 间的凹点,再逆行“爬山”得到 MT_b.因此,途中虚线箭头表示了当考虑逆行搜索时快照中需要保存的凸点.

图中的第 4 行是结合第 2 行、第 3 行所得.可以知道,只需由 BLK_{start} 到 BLK_{end} 进行一趟遍历,就可以得到上图中第 4 行这个有向图.而要得到这一行中的虚线箭头,也即快照中需要保存的凸点,其实就是在这个特殊的有向图中寻找每一个连通分支.由于这种有向图的特殊性(任意两个顶点间只可能存在一条无向路径),连通分支

的查找可以在线性时间内完成.

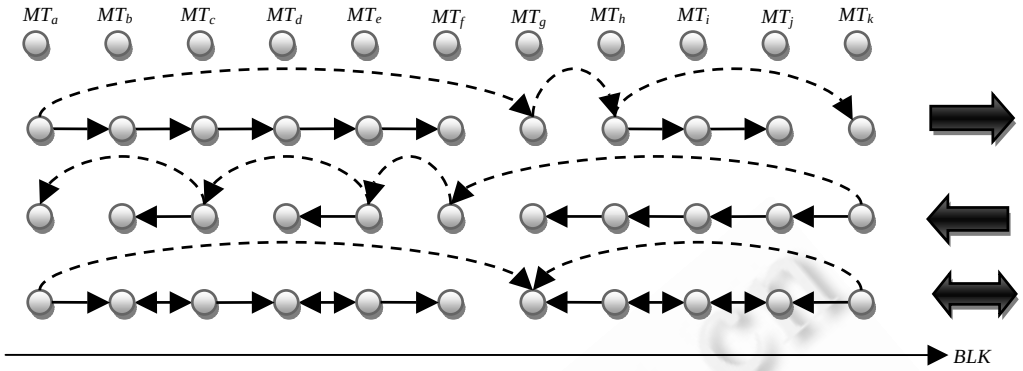


Fig.3 Optimized convex point set by retro-cost

图 3 使用路径逆行代价优化凸点集

假设预先设置路径逆行代价为 *threshold*,在累计了各个凸点的上下路线逆行代价之后,可以使用下面的算法储存快照:

atprev←1
MT_s←*nil*
prevcount←0
nextcount←0
foreach MT_i **in** $CONVEX_k$ **in** *blockno-order* **do**
  if *atprev*=0 **then**
    if *prevpathcost*(MT_i)≤*threshold* **then**
      nextcount←*nextcount*+1
      continue
    else
      atprev←1
      saveconvex(MT_s,*prevcount*,*nextcount*)
      prevcount←0
      nextcount←0
  if *atprev*=1 **then**
    MT_s←MT_i
    if *nextcount*(MT_i)≤*threshold* **then**
      prevcount←*prevcount*+1
    else
      atprev←0
    if *atprev*=1 **then**
      prevcount←*prevcount*−1
    saveconvex(MT_s,*prevcount*,*nextcount*)

为了说明方便,这种算法要求按照凸点的块号大小顺序遍历每一个凸点,对于按照平衡二叉树存储的凸点集,完全可以在线性时间内完成.算法中,*prevcount* 和 *nextcount* 分别表示被保存凸点的上下各忽略了几个凸点,这样在恢复快照的时候就可以知道逆行搜索应该在何时终止.

恢复 t 时刻的快照,与不使用逆行代价时类似,也是重建 t 时刻的 *BLKTAB* 中的 *last* 列到一个临时表 *TAB*,*TAB* 中的每一个块的索引指针首先被初始化为 *nil*:

1. 对快照中的每一个凸点执行以下步骤 2~步骤 6;
2. 将该凸点的值加入 *TAB*;
3. 上行查找,将途经的每个 MT_i 加入 *TAB*;
4. 继续向上逆行搜索,将途经的每个 MT_i 加入 *TAB*,直到发现 $prevcount(MT_i)$ 个凸点,停止在最后一个凸点上方的凹点;
5. 下行查找.将途经的每个 MT_i 加入 *TAB*;
6. 继续向下逆行搜索,将途经的每个 MT_i 加入 *TAB*,直到发现 $nextcount(MT_i)$ 个凸点,停止在最后一个凸点下方的凹点.

3 实 验

为了说明使用了 CSNAP 快照方法的 CDP 系统的空间效率,我们使用各类数据对 CSNAP 方法以及改进后的 CSNAP 方法进行模拟实验.

3.1 实验数据和方法

我们将实验分为两部分:第 1 部分使用在实际工作负载中捕获的数据,用于测试 CSNAP 在典型工作负载下的表现情况;第 2 部分使用人工合成的随机负载数据,测试在极端条件下 CSNAP 的表现情况.第 1 部分的实验数据来自两个金融行业的在线事务处理系统,样本的基本数据总结在表 1 中.因为本次实验主要关注空间效率,因此在实验中省略了每个请求的到达和处理时间;并且由于每个样本实际上包含了 10 个以上的 LUN 数据,我们本次只是从每个样本中取出了请求数量最多的 LUN 进行实验.

Table 1 Sample characteristic

表 1 样本特征数据表

	Sample 1(Financial 1)	Sample 2(Financial 2)
Number of write requests	394 562	131 868
Number of write and read requests	779 517	943 276
Percentage of write requests (%)	50.62	13.98
Number of write blocks	2 138 512	312 390
Coverage of write blocks	306 466	50 978
Minimum block number of write blocks	15	107 426
Maximum block number of write blocks	888 266	1 192 069
Average number of writes on single block	6.977975	6.127938
Maximum number of writes on single block	31 782	13 642
Average length of write requests	5.42	2.37

实验中,我们忽略 Trace 中的每一个可能的读请求.对于 Trace 中的每一个写请求,根据其中每个写入块更新 *BLKTAB* 状态表,并且保存数据块对应的元数据结构;同时,使用一个关联数组维护凸点集合.在处理完每一个写请求之后,计算并更新该时刻的凸点集合,并且累计路径逆行代价.在计算量较大的情况下,每隔固定个数的写请求计算一次凸点集合及路径逆行代价.根据凸点集合及路径逆行代价的计算结果和实验指定的阈值,我们得到需要保存到快照中的凸点数随写请求增长的变化情况,也即随时间的变化情况.

为了对两个样本的分布情况有一个直观的认识,我们用直方图显示出两个样本的分布情况.图 4 中的 x 轴表示写请求的编号,y 轴表示写请求的起始块号,z 轴表示写请求的长度,同时也使用了灰度表示写请求的长度.从图中可以看出:样本 2 相对于样本 1,写请求在 LBA 地址空间上更为分散.同时,两个样本除了少数请求长度较长以外,其他请求长度相对较短.

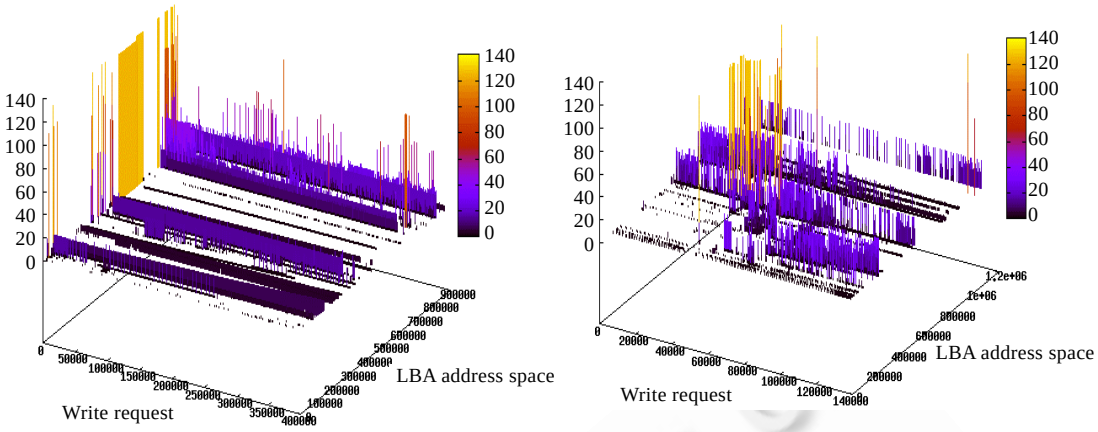


Fig.4 Distribution of write requests in sample 1 and sample 2

图 4 样本 1、样本 2 写请求分布图

3.2 实验结果分析

我们将第 1 部分实验中两个样本的结果分别总结在下面两个图表中.其中,横轴表示写请求个数,纵轴表示需要保存在快照中的凸点数.在样本 1 的图表中,纵轴使用了对数标度.图表中使用不同的曲线表示不同阈值情况下需要保存的凸点数的变化情况.其中,最上方的曲线表示阈值为 0 时,即使用基本 CSNAP 方法时的凸点变化情况,其下的曲线为依次增大阈值后所得结果.

需要指出的是,在传统的 CDP 快照方法中,快照中始终需要保存整个映射状态表 *BLKTAB*,因此在这两个样本中,我们可以认为 *BLKTAB* 的索引范围至少应该是样本中最大写入块号与最小写入块号之间的范围.因此在样本 1 中,这个范围是 888266−15=888251,而在样本 2 中,此范围是 1192069−107426=1084643.而 CSNAP 方法只需保存凸点的映射关系,而凸点个数是随着写入请求的增加而发生变化的.

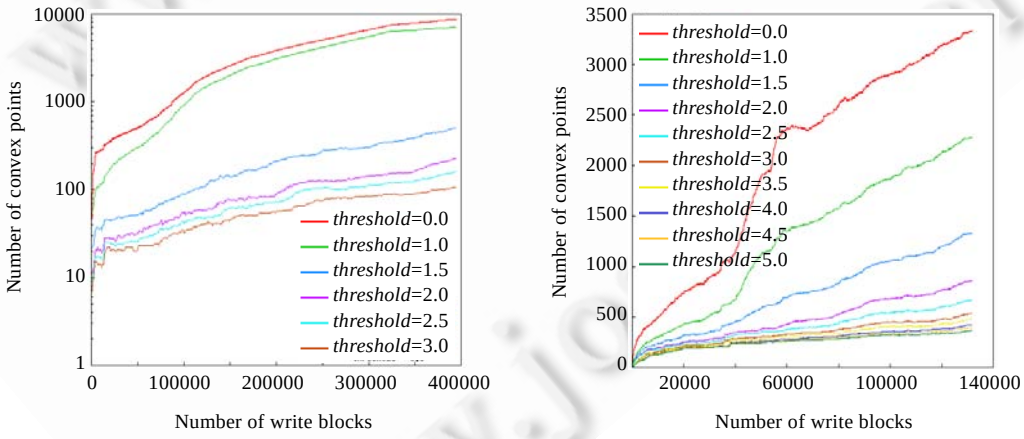


Fig.5 Number of CSNAP convex points of sample 1 and sample 2

图 5 样本 1、样本 2 下 CSNAP 方法需保存的凸点数

可以看出:在样本 1 的实验中,基本 CSNAP 方法的凸点数在写请求数接近 400 000 时,仍然在 10 000 以下,实际数值为 8 608,小于传统快照方法的 1%;而在样本 2 的实验中,在写请求数接近 140 000 时,基本 CSNAP 方法的凸点数在 3 500 以下,实际数值为 3 337,小于传统快照方法的 0.3%.即使假设传统快照方法中仅在快照中记

录状态表 *BLKTAB* 中目前为止修改过的块号,在样本 1 中仍然需要记录 306 466 个块的映射关系,而在样本 2 中也需要记录 50 978 项.使用 CSNAP 方法仍然只占这个数值的 3%和 7%.

当我们使用改进的 CSNAP 方法时,在不同的阈值条件下,需要保存的凸点数进一步减少.由前文对改进 CSANP 方法的描述中可知,路径逆行代价的最小值为 1,也就是说,这只会出现在逆行搜索每次只需要进行一次的最好情况下.从样本 1 的结果中可以看到:当设定阈值为 1 时,凸点数并没有大幅减少,在最后一个写请求后到达 7 037;而当阈值放大到 1.5 时,需要保存的凸点数出现明显的下降,在最后一个写请求之后只剩下 505 个需要保存的凸点.这说明,在样本 1 中,虽然理想情况下的逆行路径较少,但大部分逆行路径的复杂度都很小.造成这种情况的原因是,在样本 1 中,大部分的重复写操作都集中在较小的 LBA 地址范围内,并且在单位时间内的重复写入比较均匀地分布在该范围内.当阈值进一步放大时,可以看出,需要保存的凸点数减少的趋势明显放缓,也说明了在样本 1 中路径逆行代价数值分布的情况.

在样本 2 中,可以发现情况与样本 1 中完全不同.在不同的阈值条件下,需要保存的凸点数比较均匀地下降,并没有出现如样本 1 中跳跃的情况.从样本 2 的分布图中也可以看出,样本 2 的写请求分布更加分散,因此导致逆行路径的复杂度较高,各种逆行代价数值分布得比较均匀,没有出现像样本 1 那样大量集中在某个阈值范围内的情况.

由这两个样本的实验结果我们可以发现,CSNAP 方法在重复写操作相对集中的情况下表现得更好,而与重复写入的次数关系不大.结果图表中反映出,随着写请求的不断增加,需要保存的凸点数也不断增多.虽然两个样本的结果中直到最后一个写请求这个数值相对于传统快照方法仍然非常小,但我们仍然希望知道在极端条件下 CSNAP 方法的表现如何.

根据前文中的事实 2 可知,CSNAP 方法的理论最差情况是凸点与凹点相互间隔,此时凸点数恰好是整个被保护 LBA 地址空间的一半.也就是说,在理论最差情况下,CSNAP 方法占用的储存空间为传统快照方法的一半.由上面的实验结果我们看到,这种极端情况在实际工作负载下是不容易出现的,特别是在写请求分布比较集中的情况下.因此,我们使用随机数据继续进行实验.在下面的实验中,我们分别在一个大小为 1 024 个块的 LBA 地址空间中随机均匀地发起 131 072 个不同长度的写请求,观察凸点集合的退化情况.简单 CSNAP 方法的结果在图 6 中给出.其中,图 6(a)中每次实验中的写请求的长度固定,图 6(b)中每次写请求由不同长度的写请求均匀混合而成.

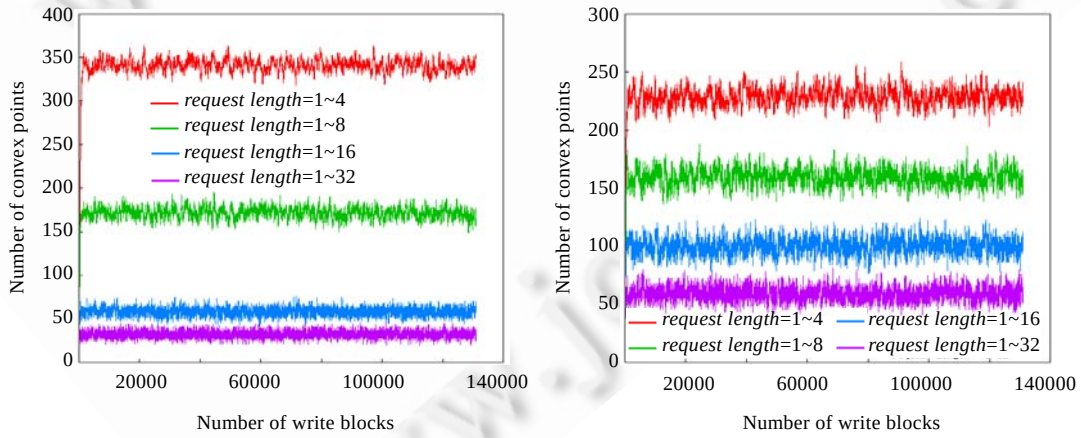


Fig.6 Number of CSNAP convex points of random workloads

图 6 随机合成负载下基本 CSNAP 方法需保存的凸点数

从图中可以看出:在随机情况下,凸点数很快稳定在一个固定数值附近;写请求长度越短,CSNAP 方法表现

得越差;但最差情况的凸点数量并没有达到理论最差情况的 512,而是接近 LBA 空间大小的 1/3.并且与预测一致,在写请求长度增大后,凸点数迅速下降.当长度达到 16 个块时,凸点数已经接近 LBA 空间的 1/20.而在写请求长度混合的实验中可以看出,结果是固定长度时各种长度的平均.

最后,我们测试在随机情况下改进 CSNAP 方法的表现(如图 7 所示).实验中,我们使用在基本 CSNAP 实验中表现最差的数据,即长度为 1 的写请求进行实验.从图中可以看出,当阈值为 2 时,需要保存的凸点数已经小于 LBA 地址空间的 1/10.

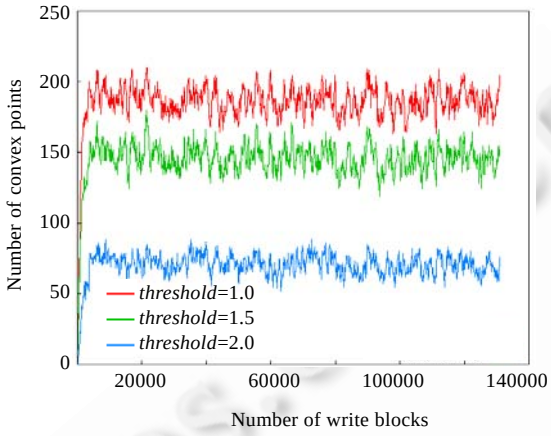


Fig.7 Number of improved-CSNAP convex points of random workloads

图 7 随机合成负载下改进 CSNAP 方法需保存的凸点数

4 CSNAP 方法的 I/O 性能

从文中对 CSNAP 方法的分析可以看出,CSNAP 方法在 I/O 性能方面存在两方面的限制:第一,需要在写入块的元数据中加入额外的字段用来维护各个写入块之间的链接关系,在更新前向链接时将产生非连续 I/O 操作;第二,在恢复快照时需要沿着元数据中保存的链接搜索,这将产生非连续的 I/O 操作,因此在记录和恢复快照时的时间效率可能低于传统快照方法.

对于第 1 个问题,非连续的 I/O 主要是由于 CSNAP 方法需要维护每一个写入块的前向链接,而前向链接的作用是用来确定凹点的位置.如果需要,可以使基本 CSNAP 方法同时记录凸点和凹点,于是,数据块元数据中就没有必要再记录前向链接,在记录 CDP 日志时将只产生连续 I/O 操作.这样,CSNAP 方法在记录 CDP 日志时就与传统快照方法相同.而由事实 2 可知,同时记录凸点和凹点的快照大小,恰好是凸点快照的一倍,根据上文实验结果,这种快照的大小仍然可以保持在传统快照的 1/10 以下.另一方面,即使 CSNAP 方法产生了非连续的 I/O 操作,通过使用预读技术及固态盘,也能有效地提高 CSNAP 方法在记录 CDP 日志时的 I/O 性能,完全可以达到大型存储系统的数据保护要求,这一点在下面的测试数据中可以看出.

对于第 2 个问题,这种情况通常只产生于高度随机的合成负载中,而且可以通过对凸点按在磁盘上的储存位置进行排序,从而产生接近连续的 I/O 操作,配合缓存使恢复快照的性能接近于传统快照方法.在实际工作负载中,对被保护磁盘的修改相对集中而不是平均分布在整个 LBA 地址空间中,沿凸点的搜索路径上的元数据结构也趋于连续.此时,CSNAP 方法由于只保存了凸点数据,在恢复快照的时间上就会优于传统快照方法.另外,由于快照本身的减小,CSNAP 方法在保存快照时对被保护磁盘 I/O 的影响也相应地减少.因为 CDP 系统在写入快照时需要阻塞对被保护磁盘的 I/O 操作,而过大的快照会花费更多的时间将快照数据写入磁盘,从而影响被保护磁盘的 I/O 性能.

下面我们通过具体数据说明 CSNAP 方法的 I/O 性能.我们分别使用上文中的实际工作负载样本 1 和合成负载进行测试,其中,合成负载在 64GB 的 LBA 空间上随机产生 419 4116 个长度为 8 个块的写请求,以测试

CSNAP方法在极端条件下的I/O性能.由于CSNAP方法和传统方法的区别主要体现在对元数据的处理上,我们这里只进行处理元数据时I/O性能的对比.实验中的CDP元数据存储使用128GB SATAII固态盘.测试分别统计处理完所有I/O、保存快照和恢复快照所用时间.

我们将测试的结果总结在表2中.从中可以看出,在实际工作负载中,CSNAP方法只在记录CDP日志时稍慢于快照传统方法,而在保存和恢复快照时明显快于传统方法.这主要是由于,在实际负载中,对被保护磁盘的写操作多集中在较小的LBA范围内,因而快照中需要保存的凸点数较少;而传统快照方法需要将所有LBA的映射关系保存在快照中.

Table 2 I/O performance of CSNAP

表2 CSNAP方法的I/O性能

	Sample 1(Financial 1)	Synthetic workload
Time of processing all I/Os (traditional method) (s)	10.218 3	13.950 3
Time of taking snapshot (traditional method) (s)	16.064 1	16.515 3
Time of restoring snapshot (traditional method) (s)	18.526 9	19.046 0
Time of processing all I/Os (CSNAP method) (s)	13.307 0	484.957 3
Time of taking snapshot (CSNAP method) (s)	0.028 6	3.027 0
Time of restoring snapshot (CSNAP method) (s)	2.493 5	19.342 9

另一方面,在极端条件下,CSNAP方法需要保存的凸点数增加,所以保存快照的时间相对于实际工作负载时有所增加,但仍然明显快于传统快照方法.同时,在恢复快照时,由于整个LBA空间被均匀而完全地写入,CSNAP方法需要遍历的元数据也相应增加,到达极端最差情况.在此情况下,CSNAP方法和传统快照方法需要相同的时间恢复快照.在记录CDP日志时可以看出,CSNAP方法相对于实际负载时间出现了明显延长,这是由于随机负载导致了在更新元数据中的前向链接时出现了非连续I/O.如前文所说,这里可以取消前向链接字段,而同时保存凸点和凹点,从而使记录CDP日志的时间和传统快照方法达到一致.而保存快照的时间也仅增加1倍,变为6s,仍然明显快于传统快照方法,而恢复快照的时间保持不变.

实际上,只要我们计算一下I/Ops就可以知道,CSNAP方法即使在合成负载的极端条件下也已经足够满足大型储存系统的需要,而不必特别取消前向链接字段.因为合成负载一共产生了4 194 116个写请求,而CSNAP处理这些请求用时484.957 3s,处理速度已经达到8 648IOps.而我们从对IBM储存系统的测试数据表格[14]中可以看到,即使是使用80个磁盘的大型存储系统,其可能产生的工作负载最大也只有8 170IOps.对于更大的存储系统,只要将CDP日志记录在多盘RAID 0阵列上,就可以使CSNAP方法的I/O处理速度进一步得以提高.

5 结 论

在本文中,我们提出了一种在连续数据保护系统中利用凸点集合保存快照的方法,即CSNAP方法,并且提出逆行代价的概念对CSNAP方法进行改进.通过使用实际工作负载和随机工作负载数据进行实验,CSNAP方法在实际情形下占用的空间小于传统方法的10%,而改进的CSNAP方法甚至小于传统方法的1%.而在随机数据的最差情况下,虽然简单CSNAP方法占用空间增长到传统方法的33%,但改进的CSNAP方法仍然将需要保存的凸点数控制在传统方法的10%以下.由于需要储存的数据量的减少,使得储存快照的时间也相应地缩短,因此,当储存快照时,对正在运行的系统的性能产生的影响也比使用传统快照方法时要小.

关于CSNAP方法的I/O性能,如上节所述,与传统快照方法相比,特别是在实际工作负载中,保存和恢复快照都有着明显的优势.而记录CDP日志的处理能力即使是在极端条件的合成负载下,也能够满足大型存储系统的应用要求.对于在元数据结构中增加的字段,只要与储存快照时节省的储存空间比较,在数据块元数据中增加的字段就是可以接受的了.例如,对一个1 000GB的储存系统进行连续数据保护,假设该系统的吞吐量为2 000 IOps,那么如果使用CSNAP方法每天将产生约5GB的数据块元数据,而传统快照方法产生约1.3GB数据块元数据.但传统方法的快照需要约15GB,而使用改进的CSNAP方法,我们可以期待150MB的快照大小.因为这150MB的数据块元数据可能分散在磁盘上不连续的区域中,因此恢复快照的速度可能受到限制.但是,如果将数据块元数据储存在固态盘上,那么这种随机型I/O的性能也能够达到传统硬盘连续型I/O的性能.

因此,通过上面的分析可知,CSNAP 方法相对于现有方法可以更为高效地在 CDP 系统中储存快照,并且在储存快照时减少对系统带来的影响.

References:

[1] Wallis J. Common causes for data loss. 2008. http://www.isnare.com/?aid=222505&ca=Computers+and+Technology

[2] Wang SP, Yun XC, Guo L. Continuous data protection (CDP) technology overview. Information Technology Letter,2008,6(6): 24−33 (in Chinese with English abstract).

[3] Xiao WJ, Yang Q, Ren J. A case for continuous data protection at block level in disk array storages. In: Proc. of the IEEE Trans. on Parallel and Distributed Systems. IEEE Computer Society, 2009. 898−911. [doi: 10.1109/TPDS.2008.154]

[4] Lin SB, Chiueh TC, Lu MH. An incremental file system consistency checker for block-level CDP systems. In: Proc. of the IEEE Symp. on Reliable Distributed Systems. IEEE Computer Society, 2008. 157−162. [doi: 10.1109/SRDS.2008.20]

[5] Lu MH, Chiueh TC, Lin SB. Efficient logging and replication techniques for comprehensive data protection. In: Proc. of the IEEE/NASA Goddard Conf. on Mass Storage Systems and Technologies. IEEE Computer Society, 2007. 171−184. [doi: 10.1109/MSST.2007.14]

[6] Lu MH, Chiueh TC. File versioning for block-level continuous data protection. In: Proc. of the Int'l Conf. on Distributed Computing Systems. IEEE Computer Society, 2009. 327−334. [doi: 10.1109/ICDCS.2009.48]

[7] Li X, Xie CS, Yang Q. Optimal implementation of continuous data protection (CDP) in Linux kernel. In: Proc. of the Int'l Conf. on Networking, Architecture, and Storage. IEEE Computer Society, 2008. 28−35. [doi: 10.1109/NAS.2008.44]

[8] Chiueh TC, Zhu NN. Portable and efficient continuous data protection for network file servers. In: Proc. of the Int'l Conf. on Dependable Systems and Networks. IEEE Computer Society, 2007. 687−697. [doi: 10.1109/DSN.2007.74]

[9] Brinkmann A, Effert S. Snapshots and continuous data replication in cluster storage environments. In: Proc. of the IEEE Int'l Workshop on Storage Network Architecture and Parallel I/Os. IEEE Computer Society, 2008. 3−10. [doi: 10.1109/SNAPI.2007.13]

[10] Ju DP, Wang DS, He JY, Sheng YH. TH-CDP: An efficient block level continuous data protection system. In: Proc. of the Int'l Conf. on Networking, Architecture, and Storage. IEEE Computer Society, 2009. 395−404. [doi: 10.1109/NAS.2009.69]

[11] Wu GJ, Yun XC, Fang BX, Wang SP, Yu XZ. HCSIM: An indexing method for long-lived frequent block-level snapshot. Chinese Journal of Computers, 2009,32(10):2080−2090 (in Chinese with English abstract).

[12] Ren J, Xiao WJ, Yang Q. TRAP-Array: A disk array architecture providing timely recovery to any point-in-time. In: Proc. of the Int'l Symp. on Computer Architecture. IEEE Computer Society, 2006. 289−301. [doi: 10.1109/ISCA.2006.44]

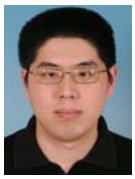
[13] Li ZH, Zhou K, Yang TM, Liu JN. TSPSCDP: A time-stamp continuous data protection approach based on pipeline strategy. In: Proc. of the Japan-China Joint Workshop on Frontier of Computer Science and Technology. IEEE Computer Society, 2008. 96−102. [doi: 10.1109/FCST.2008.27]

[14] Stephan C. Comparison of the performance of the DS4300 (FAStT600) and FAStT200 storage servers. 2004. http://www-03.ibm.com/systems/x/resources/benchmarks/whitepapers/

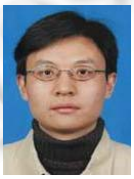
附中文参考文献:

[2] 王树鹏,云晓春,郭莉.持续数据保护(CDP)技术的发展综述.信息技术快报,2008,6(6):24−33.

[11] 吴广君,云晓春,方滨兴,王树鹏,余翔湛.HCSIM:一种长期高频 Block-Level 快照索引技术.计算机学报,2009,32(10):2080−2090.



李琥(1982－),男,北京人,博士生,CCF 学生会员,主要研究领域为计算机体系结构,计算机存储.



李元章(1978－),男,博士生,讲师,主要研究领域为网络存储,嵌入式系统.

谭毓安(1972－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为网络存储,计算机信息安全,嵌入式系统.